How Would You Decide?

What's the Problem?
My sister socked me in the nose.

Information Needed
My sister is J. Hitler, it hurts.

Decision Options

Option A
Punch her lightly
- Positive Consequences: I will be happy
- Negative Consequences: I will get in trouble
- Factors Influencing Me

Option B
Tell on her
- Positive Consequences: She will get in trouble
- Negative Consequences: I will be a troublemaker

Option C
Lock her in the closet
- Positive Consequences: I will never see her again
- Negative Consequences: I don't see any

My Choice: C
We will (finally) move away from uncertainty (for a bit) and instead focus on learning.

Learning algorithms benefit from flexibility to solve a wide range of problems, especially:

1. Cannot explicitly program (what set of if-statements/loops tells dogs from cats?)

2. Answers might change over time (what is “trendy” right now?)
Learning

We can categorize learning into three types:

Unsupervised = No explicit feedback

Reinforcement = Get a reward or penalty based on quality of answer

Supervised = Have a set of inputs with the correct answer/output ("labeled data")
Learning

Horse:  

Unsupervised: 
Computer guesses: “donkey”, “spaceship”
You don’t tell it anything: “...”

House:
Learning

Horse: 

House: 

Reinforcement:

Computer guesses: “donkey”, “spaceship”

You answer: “close”, “not really”
Learning

Horse:  

Supervised: 

Computer doesn’t guess: “...” 

You tell: “that is horse”, “that is house”
We can categorize learning into three types:

Unsupervised = No explicit feedback

Reinforcement = Get a reward or penalty based on quality of answer

easiest... so we will assume this for a while

Supervised = Have a set of inputs with the correct answer/output ("labeled data")
Learning Trade-offs

One import rule is **Ockham’s razor** which is: if two options work equally well, pick simpler.

For example, assume we want to find/learn a line that passes through: (0,0), (1,1), (2,2).

Quite obviously “y=x” works, but so does “y=x^3-3x^2+3x” … “y=x” is a better choice.
A similar (but not same) issue that we often face in learning is **overfitting**.

This is when you try too hard to match your data and lose a picture of the “general” pattern.

This is especially important if noise or errors are present in the data we use to learn (called **training data**).
Learning Trade-offs

A simple example is suppose you want a line that passes through more points: (0,0), (1,1), (2,2), (3,3), (4,4), (5,5.1), (6,6)

Line “y=x” does not quite work due to (5,5.1)

But it might not be worth using a degree 6 polynomial (not because finding one is hard), as it will “wiggle” a lot, so if we asked for y when x=10... it will be huge (or very negative)
One of the simplest ways of learning is a decision tree (i.e. a flowchart... but no loops)

For example, you could classify movies as:
Decision Trees

One of the simplest ways of learning is a **decision tree** (i.e. a flowchart... but no loops)

For example, you could classify movies as:

- **violent?**
  - yes: **historical?**
    - yes: war
    - no: action
  - no: **love?**
    - yes: romance
    - no: funny?
      - yes: comedy
      - no: family

**call these attributes/inputs**

**outputs/classification**
Decision Trees

If I wanted to classify Deadpool our inputs might be:
[vViolent=yes, historical=no, love=not really, funny=yes]
Decision Trees

In our previous example, the attributes/inputs were binary (T/F) and output multivariate.

The math is it simpler the other way around, input=multivariate & output=binary.

An example of this might be deciding on whether or not you should start your homework early or not.
Decision Trees

Do homework early example:

- **when assigned?**
  - over 1 week ago
  - less than a week
    - yes
      - **number of problems?**
        - <3
          - aww; no
        - 3 to 5
        - >5
          - yes
            - understand topic?
              - yes
              - no
            - sorta
            - back of hand
          - yes
          - no
        - not really
Making Trees

... but how do you **make** a tree from data?

<table>
<thead>
<tr>
<th>Example</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>Ans</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>low</td>
<td>big</td>
<td>twit</td>
<td>5</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>low</td>
<td>small</td>
<td>FB</td>
<td>8</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>med</td>
<td>small</td>
<td>FB</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>high</td>
<td>big</td>
<td>snap</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>high</td>
<td>small</td>
<td>goog</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>med</td>
<td>big</td>
<td>snap</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>low</td>
<td>big</td>
<td>goog</td>
<td>9</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>high</td>
<td>big</td>
<td>goog</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>med</td>
<td>small</td>
<td>twit</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>high</td>
<td>small</td>
<td>goog</td>
<td>4</td>
<td>F</td>
</tr>
</tbody>
</table>
Making Tress: Brute Force

The brute force (try every option; find best) way would be: let \( n = 5 = \) number attributes

If these were all T/F attributes... there would be \( 2^n = 2^5 \) rows for a full truth table

<table>
<thead>
<tr>
<th>Example</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>Ans</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>low</td>
<td>big</td>
<td>twit</td>
<td>5</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>low</td>
<td>small</td>
<td>FB</td>
<td>8</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>med</td>
<td>small</td>
<td>FB</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>high</td>
<td>big</td>
<td>snap</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>high</td>
<td>small</td>
<td>goog</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>med</td>
<td>big</td>
<td>snap</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>low</td>
<td>big</td>
<td>goog</td>
<td>9</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>high</td>
<td>big</td>
<td>goog</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>med</td>
<td>small</td>
<td>twit</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>high</td>
<td>small</td>
<td>goog</td>
<td>4</td>
<td>F</td>
</tr>
</tbody>
</table>
Making Tress: Brute Force

But each row of the truth table could be T/F

So the number of T/F combinations in the answer is:

\[ 2^{rows} = 2^{2^n} \]

This is very gross, so brute force is out.
Making Tress: Recursive

There are two key facts to notice:
(1) You need to pick an attribute to “split” on
(2) Then you have a recursive problem
(1 less attribute, fewer examples)
Making Tress: Recursive

This gives a fairly straight-forward recursive algorithm:

def makeTree(examples):
    if output all T (or all F), make a leaf & stop
    else (1) A=pick attribute to split on
         for all values of A:
             (2) makeTree(examples with A val)
Making Tress: Recursive

What attribute should you split on?

Does it matter?

If so, what properties do you want?
What attribute should you split on?
A very difficult question, the best answer is intractable so we will approximate

Does it matter?
Yes, quite a bit!

If so, what properties do you want?
We want a variable that separates the trues from falses as much as possible
To determine which node to use, we will do what CSci people are best at: copy-paste someone else’s hard work.

Specifically, we will “borrow” ideas from information theory about entropy (which, in turn, is a term information theory “borrowed” from physics).

Entropy means a measure of disorder/chaos.
Entropy

You can think of entropy as the number of “bits” needed to represent a problem/outcome.

For example, if you flipped a fair coin... you get heads/tails 50/50.

You need to remember both numbers (equally) so you need 1 bit (0 or 1) for both possibilities.
Entropy

If you rolled a 4-sided die, you would need to remember 4 numbers (1, 2, 3, 4) = 2 bits

A 6-sided die would be $\log_2(6) = 2.585$ bits

If the probabilities are not uniform, the system is less chaotic... (fewer bits to “store” results)

So a coin always lands heads up: $\log_2(1) = 0$
Entropy

Since a 50/50 coin = 1 entropy/bits
... and a 100/0 coin = 0 entropy/bits

Then a 80/20 coin = between 0 and 1 bits

The formal formula is entropy, $H(V)$, is:

$$H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = - \sum_k P(v_k) \log_2 P(v_k)$$

... where $V$ is a random variable and $v_k$ is one entry in $V$ (only uses prob, not value part)
Entropy

\[ H(V) = \sum_{k} P(v_k) \log_2 \frac{1}{P(v_k)} = - \sum_{k} P(v_k) \log_2 P(v_k) \]

... so a 50/50 coin is random variable:
\[ x = [(0.5, \text{heads}), (0.5, \text{tails})] \]
\[ H(x) = -0.5 \cdot \log_2(0.5) - 0.5 \cdot \log_2(0.5) = 1 \]

Then... for our other examples:
\[ y = [(0.8, \text{heads}), (0.2, \text{tails})] \]
\[ H(y) = -0.8 \cdot \log_2(0.8) - 0.2 \cdot \log_2(0.2) = 0.7219 \]

\[ z = [(1/6, 1), (1/6, 2), (1/6, 3), \ldots (1/6, 6)] \]
\[ H(z) = 6 \cdot (-\frac{1}{6} \cdot \log_2(\frac{1}{6})) \]
\[ = - \log_2(\frac{1}{6}) = \log_2(6) = 2.585 \]
How can we use entropy to find good splits?

\[ H(V) = \sum_{k} P(v_k) \log_2 \frac{1}{P(v_k)} = - \sum_{k} P(v_k) \log_2 P(v_k) \]
Entropy

How can we use entropy to find good splits?

$$H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k)$$

Compare entropy/disorder before and after split:

<table>
<thead>
<tr>
<th>Example</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>Ans</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>low</td>
<td>big</td>
<td>twit</td>
<td>5</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>low</td>
<td>small</td>
<td>FB</td>
<td>8</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>med</td>
<td>small</td>
<td>FB</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>high</td>
<td>big</td>
<td>snap</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>high</td>
<td>small</td>
<td>goog</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>med</td>
<td>big</td>
<td>snap</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>low</td>
<td>big</td>
<td>goog</td>
<td>9</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>high</td>
<td>big</td>
<td>goog</td>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>med</td>
<td>small</td>
<td>twit</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>high</td>
<td>small</td>
<td>goog</td>
<td>4</td>
<td>F</td>
</tr>
</tbody>
</table>

Before: 5 T, 5 F

Split A

4 T, 2F

1 T, 3 F
Entropy

How can we use entropy to find good splits?

\[ H(V) = \sum_{k} P(v_k) \log_2 \frac{1}{P(v_k)} = - \sum_{k} P(v_k) \log_2 P(v_k) \]

Compare entropy/disorder before and after split:

- % of total true

\[ H(before) = -0.5 \cdot \log_2(0.5) - 0.5 \cdot \log_2(0.5) = 1 \]

\[ H(after = T) = -0.667 \cdot \log_2(0.667) - 0.333 \cdot \log_2(0.333) = 0.918 \]

\[ H(after = F) = -0.25 \cdot \log_2(0.25) - 0.75 \cdot \log_2(0.75) = 0.811 \]
How can we use entropy to find good splits?

\[ H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k) \]

Compare entropy/disorder before and after split:

<table>
<thead>
<tr>
<th>% of total true</th>
<th>5 T, 5 F</th>
<th>how combine?</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>4 T, 2 F</td>
<td>1 T, 3 F</td>
</tr>
</tbody>
</table>

\[
H(\text{before}) = -0.5 \cdot \log_2(0.5) - 0.5 \cdot \log_2(0.5) = 1
\]

\[
H(\text{after} = T) = -0.667 \cdot \log_2(0.667) - 0.333 \cdot \log_2(0.333) = 0.918
\]

\[
H(\text{after} = F) = -0.25 \cdot \log_2(0.25) - 0.75 \cdot \log_2(0.75) = 0.811
\]
Entropy

Random variables (of course)!

\[ A_{after} = [(6/10, 0.918), (4/10, 0.811)] \]

6 of 10 examples had \( A = T \)

So expected/average entropy after is:

\[ E[A_{after}] = 0.6 \cdot 0.918 + 0.4 \cdot 0.811 = 0.875 \]

We can then compute the difference (or gain):

\[ Gain(A) = H(before) - H(A_{after}) = 1 - 0.875 = 0.125 \]

More “gain” is means less disorder after
So we can find the “gain” for each attribute and pick the argmax attribute. This greedy approach is not guaranteed to get the shallowest (best) tree, but does well. However, we might be over-fitting the data... but we can use entropy also determine this. 
Next we will do some statistics

\rantOn
Statistics is great at helping you make correct/accurate results

Consider this runtime data, is alg. A better?

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5.2</td>
<td>6.4</td>
<td>3.5</td>
<td>4.8</td>
</tr>
<tr>
<td>B</td>
<td>5.8</td>
<td>7.0</td>
<td>2.8</td>
<td>5.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.0</td>
</tr>
</tbody>
</table>
Statistics Rant

Not really... only a 20.31% chance A is better (too few samples, difference small, var large)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5.2</td>
<td>6.4</td>
<td>3.5</td>
<td>4.8</td>
<td>3.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5.8</td>
<td>7.0</td>
<td>2.8</td>
<td>5.1</td>
<td>4.0</td>
</tr>
</tbody>
</table>

Yet, A is faster 80% of the time... so you might be mislead in how great you think your algorithm is
We can frame the problem as: what is the probability that this attribute just randomly classifies the result.

Before our “A” split, we had with 5T and 5F. A=T had 4T and 2F.

So 6/10 of our examples went A=T... if these 6/10 randomly picked from the 5T/5F, we should get 5*6/10 T on average randomly.
Decision Tree Pruning

Formally, let \( p = \text{before T} = 5 \), \( n = \text{before false} = 5 \)

\[
p_{A=T} = T \quad \text{when } \text{“A=T”} = 4
\]

\[
n_{A=F} = F \quad \text{when } \text{“A=T”} = 2
\]

... and similarly for \( p_{A=F} \) and \( n_{A=F} \)

Then we compute the expected “random” outcomes:

\[
\hat{p}_k = p \cdot \frac{p_k + n_k}{p + n}
\]

\[
\hat{n}_k = n \cdot \frac{p_k + n_k}{p + n}
\]

\[5 \times \frac{6}{10} = 3 \text{ T on average by “luck”}\]
Decision Tree Pruning

We then compute (a “test statistic”):

\[
x = \sum_k \left( \frac{(p_k - \hat{p}_k)^2}{\hat{p}_k} + \frac{(n_k - \hat{n}_k)^2}{\hat{n}_k} \right)
\]

\[
= \frac{(p_{A=T} - \hat{p}_{A=T})^2}{\hat{p}_{A=T}} + \frac{(n_{A=T} - \hat{n}_{A=T})^2}{\hat{n}_{A=T}}
\]

\[
+ \frac{(p_{A=F} - \hat{p}_{A=F})^2}{\hat{p}_{A=F}} + \frac{(n_{A=F} - \hat{n}_{A=F})^2}{\hat{n}_{A=F}}
\]

\[
= \frac{(4 - 3)^2}{3} + \frac{(2 - 3)^2}{3} + \frac{(1 - 2)^2}{2} + \frac{(3 - 2)^2}{2}
\]

\[
= 1.667
\]
Once we have “x” we can jam it into the $\chi^2$ (chi-squared) distribution:

$$\chi^2(1)(x) = \chi^2(1)(1.667) = 0.19671$$

So there is a 19.67% chance this variable is just “randomly” assigning... so we might want to not use “A” here (other places maybe) for T/F happens when $x>3.841$

The “typical” threshold we look for is 5% of being “random”... if so, could collapse node
What is this $\chi^2$ thing?

I think most people are familiar with the “bell”/normal/Gaussian distribution:

$P(x<2)$

$N(\mu, \sigma^2)(x)$ needs 2 parameters: $\mu, \sigma$

$$\int_{-\infty}^{x} P(z)dz = P(x < z)$$
What is this $\chi^2$ thing?

$\chi^2$ is just a different distribution that only requires 1 parameter (degrees of freedom)

Written both as $\chi^2(k,x)$ or $\chi^2(k)(x)$

a statistics thing... out of the scope of this course
Decision Tree Pruning

So, suppose you had a “bad” attribute (conflicting examples/inputs in this case):

Notice the attribute “X” is not really helping (at all...), so you could just remove it.

leaf node, ran out of attributes...

more T than F so just “guess” T

Notice the attribute “X” is not really helping (at all...), so you could just remove it.
Complications

There are a number of complications:
(1) Attributes with more possible “values” seem better than they are
(2) Integers/doubles you typically want to threshold to remove issue of (1)
(3) If you want a continuous output rather than a classification, your leaf needs to be a function rather than a single value